20010226 119

# A Parallel, Real-Time Garbage Collector

Perry Cheng
pscheng@cs.cmu.edu

Guy Blelloch
guyb@cs.cmu.edu

2000

## ABSTRACT

In an earlier paper [2], we presented an algorithm for a parallel, real-time garbage collector for shared memory multiprocessors with provable bounds on time and space usage. Since the algorithm was designed to keep the analysis simple, it had some features that were impractical. This paper presents the modifications necessary for a practical implementation: reducing excessive interleaving, handling stacks and global variables, reducing double allocation, and special treatment of large and small objects. An implementation based on the modified algorithm is evaluated on a set of 15 SML benchmarks on an UltraSparc-II multiprocessor. To our knowledge, this is the first implementation of a parallel, concurrent, real-time garbage collector.

On 8 processors, our collector has a speedup ranging from 5.5 to 7.8 and maximum pause times from 3 ms to 5 ms. In contrast, a non-incremental collector (whether generational or not) has maximum pause times from 10 ms to 650 ms. Compared to a non-parallel, stop-copy collector, parallelism has a 39% overhead, while achieving real-time behavior has an additional 12% overhead. Since the collector takes about 15% of total execution time, these features respectively have an overall time costs of 6% and 2%.

## 1. INTRODUCTION

The first garbage collectors were non-incremental, non-parallel collectors appropriate for batch workloads on uniprocessors [12, 3, 17]. However, parallel or multi-threaded applications that run on multiprocessors need parallel garbage collectors since even a dedicated processor cannot keep up with the memory requirements of more than a few processors. Early efforts include Halstead's parallel collector for MultiLISP [13] which, however, has problems scaling because it does not load-balance the collection work. More recently, Endo used load balancing to achieve a scalable mark-sweep collector [10]. However, the collector cannot bound the memory footprint because of the potential for fragmentation inherent in a non-moving collector. Other researchers tackled the problem of eliminating the lengthy pauses the application experiences during garbage collection using incremental and concurrent collection. Incremental collectors break up each garbage collection into smaller pieces of work and interleave these segments of collection work within the execution of the program [1]. Concurrent collectors run a single collector thread concurrently with one or more application threads [19, 6]. These collectors, however, do not consider running multiple collector threads in parallel.

In an earlier paper, we presented an algorithm for a scalably parallel, real-time garbage collector for shared memory multiprocessors with provable time and space bounds [2]. By making all aspects of the collector incremental (*e.g.*, incrementally copying arrays) and using appropriate synchronization constructs (*e.g.*, avoiding locks), we were able to achieve tight bounds on the pause time of the application. Using load balancing, the idle time of any garbage-collecting processor is bounded so that the collector's parallelism is scalable. This guarantees the timely completion of a collection. In the setting of a concurrent collector where the application continues to allocate during collection, timeliness is expressible as a space bound. Specifically, for any application with maximum reachable space $R$, maximum object count $N$, and maximum depth of the memory graph $D$, our collector on a $P$-way multiprocessor requires at most $2(R(1 + 1.5/k) + N + 5PD)$ space where $k$ is a parameter controlling the rate of collection. While our algorithm was theoretically sound, it had some features that were impractical. Furthermore our initial work was purely a theoretical result with no experimental analysis.

The contributions of this paper are twofold. First, we describe a set of modifications to our previous algorithm that we found necessary in developing a practical implementation. Second, we have fully implemented the algorithm as part of an existing run-time system and present performance numbers for the algorithm. The most innovative modifications include extending the algorithm to work with generations, being able to include stacks in the model by using the notion of stacklets, and reducing the cost of load-balancing using a multi-room synchronization abstraction. Other modifications include increasing the granularity of the incremental steps, separately handling global variables, delaying the copy on write, reducing the synchronization costs of copying small objects, and avoiding double allocations during collection.

To evaluate the collector algorithm, we implemented several variants within the runtime system [4] for the TILT SML compiler. This includes stop-copy and concurrent (real-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2000 ACM 0-89791-88-6/97/05 ..$5.00

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

time) variants, and semi-space and generational variants. To our knowledge, this is the first implementation of a parallel, real-time garbage collector. The implementations were evaluated on a set of 15 benchmarks on an UltraSparc-II multiprocessor. Compared to a baseline non-incremental, non-parallel collector, scalable parallelism on average has a 39% overhead while concurrency and real-time bounds add another 12%. Since the collector takes about 15% of the total execution time, these translate to an overall time cost of 6% and 2%, respectively. The speedup of the collector on 8 processors varies from 5.5 to 7.8. Finally, our maximum pause time is about 3 ms to 5 ms.

## 2. BACKGROUND AND DEFINITIONS

We describe a traditional *semispace* copying garbage collector [3], define when a collector is incremental, parallel, or concurrent, and introduce a new notion for measuring how real-time a collector is.

### 2.1 A Semispace Stop-Copy Collector

In a semispace collector, heap memory is divided into two equally-sized regions: the *fromspace* and the *tospace*. During normal execution, the mutator allocates new objects from fromspace. Eventually, continued allocation exhausts fromspace causing the program to be suspended while the collector reclaims memory. The program's data can be viewed as a graph in which the nodes are objects, the edges are pointer fields of the objects, and the roots are the pointer values of the register set. The objects that can be reached from the roots are called the *reachable* objects. Under this interpretation, the collector traverses the memory graph starting from the roots and copies the reachable objects from fromspace into tospace. When all reachable objects are copied, the collector is *flipped* off by updating the root values and reversing the roles of fromspace and tospace.

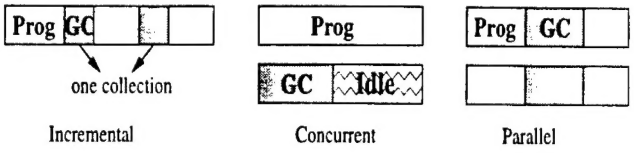
A useful terminology for describing garbage collection is the tri-color abstraction [6] which assigns one of three colors (white, gray, and black) to objects. When a collection starts, all reachable objects are white. Unreachable objects have no color and require no consideration since they will never be manipulated by either the mutator or the collector. When a white object is copied, it becomes gray, signifying that it has been copied but that it may still refer to white objects. When all of a gray object's children are copied it becomes black. A collection is complete when all objects are black. To properly update all the pointers during collection so that they point to the new copy, the collector needs to know the new location of each object. This is typically maintained with a *forwarding pointer* from the old copy to the new one.

### 2.2 Types of Garbage Collectors

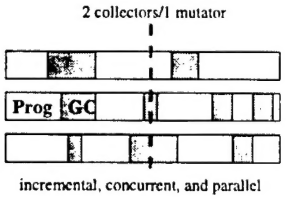
The first garbage collectors, including the one described in the previous section, were non-incremental (*stop-collect*) and programs (also called *mutators*) were periodically suspended while the collector reclaimed memory. Over the years, various techniques were introduced to address the problems of collector efficiency, reducing or eliminating pause times, and adapting a collector for use on multiprocessors. Many of these techniques or collector properties are often used in isolation to address a particular problem, creating a misperception that the techniques are mutually exclusive. In fact, these techniques sometimes complement each other. To avoid confusion, we define the following terms:

- Incremental[1]: A single collection is divided into multiple segments whose executions are interleaved with the application. Typically, the rate of collection is related to the allocation rate so that the collection is guaranteed to terminate.
- Concurrent[19, 6]: At least one program thread and one collector thread are executing concurrently.
- Parallel[13, 10]: Multiple collector threads are collecting concurrently.

The reader should be aware that these terms are not used consistently across the literature. For example, some papers refer to concurrent collectors as being parallel while others exclude the notion of incrementality when they use the term parallel. The diagram below shows one collection cycle for the 3 types of collectors for the 2-processor case:

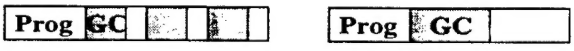



Both our original and modified collectors are simultaneously incremental, concurrent and parallel. Each processor runs incrementally, but since the processors are not synchronized, any combination of mutator and collector threads can run concurrently. This can be illustrated as follows:




### 2.3 What is a Real-time collector?

Empirical studies of real-time garbage collectors typically report pause times, with particular emphasis on the maximum pause time. What can a programmer deduce from knowing that the maximum pause time is 25ms? Pessimistically, he can deduce this is unsuitable for applications that require a response time of 10ms. However, he cannot safely assume the collector is suitable for mouse tracking, which requires a response time of about 50ms. The problem is that statistics about the pause time do not characterize the occurrences of the pauses. A burst of frequent short pauses is not much different from a single long pause.




Many short pauses  Few long pauses

To capture both the size and placement of pauses, we propose a notion of *utilization*. In any time window, we define the utilization of that window to be the fraction of time that the mutator executes. For any window size, the minimum utilization over all windows of that size captures the access the mutator has to the processor at that granularity. For example, mouse tracking might require a response time of 50ms and the mouse-tracking code might take 1ms to execute. In that case, it is sufficient to have a minimum

utilization of 2% at a granularity of 50ms. The minimum mutator utilization (MMU) is a function of window size and generalizes both maximum pause time and collector overhead. The maximum pause time is the window size at and below which the MMU is zero and the collector overhead is the complement of the MMU at the granularity of the total execution time.

# 3. ALGORITHM

In this section, we review our previous collector algorithm [2] by a series of modifications to Cheney's simple copying collector [3]. Our treatment adds, in turn, parallelism, incremental collection, and concurrency. We end by describing the space and time bounds guaranteed by the collector.

The collector is designed for a shared memory multiprocessor. In addition to the typical uniprocessor instructions, the collector uses a `FetchAndAdd` instruction, which atomically reads a memory location and stores the incremented value back to the location, and a `TestAndSet`, which atomically reads a memory location and, if the read value equals some given value, updates the same memory location with a new value. The collector interfaces with the application via 3 memory-related operations: allocating space for a new object, initializing the fields of a new objects, and modifying the field of an existing object.

## 3.1 Scalable Parallelism

Most of the work in a copying collector is in processing gray objects. A gray object is colored black after each of its white descendants (if any) are copied and colored gray. The manner in which the set of gray objects is maintained is important. Cheney proposed a clever way of maintaining the set of gray objects at no additional space cost by noting that the gray objects are contiguous in tospace if a breadth-first-order is used to traverse the memory graph[3]. This idea works only if the heap can be parsed by scanning objects in memory order, a requirement easily satisfied when objects are tagged.

Using an implicit data structure, however, has some disadvantages. First, it is difficult to experiment with other traversal orders. Second, with multiple processors, it is not possible to keep the gray objects contiguous without excessive synchronization. Related to this point is the need to manipulate sets of gray objects for purposes of load-balancing, a key idea in achieving scalable parallelism. For these reasons, our collector represents the set of gray objects using an explicitly-managed local (per-processor) stack. (A queue or other different data structure is also possible.)

For the collector to scale, it is important that no processor is idle during the collection. This can happen if one processor runs out of gray objects while collectively there are still more gray objects to be processed. Work sharing is one way of distributing the workload to avoid idleness. In addition to the local per-processor stacks, we add a shared stack of gray objects accessed by all processors. Each processor periodically transfers gray objects between its local stack and the shared stack. Thus no processor idles until there are no more gray objects in the shared stack.

The main correctness issue that arises with multiple copying collectors is the possibility that a white object is erroneously copied twice, resulting in an incorrect replica graph. To prevent this from occurring, a per-object synchronization is used for object copying. Before an object is copied, a copier must gain exclusive access to the object by atomically swapping into the forwarding pointer field a flag designating that copying is in progress. Other processors, if any, that had desired to copy the object would have failed to perform the atomic swap and instead noticed that copying is in progress. These processors busy-wait until the forwarding pointer is installed. It is important that the forwarding pointer can be installed before the fields of the object are copied since this guarantees that the wait, if any, is at most a handful of cycles. We refer to this mechanism as the *copy-copy synchronization.*

Our shared stack is based on the observation that multiple stack pushes or multiple stack pops interfere with each other less seriously than a push and a pop. In fact, multiple pushes without wait-locks can be achieved by using a `FetchAndAdd` to reserve a target region in the shared stack prior to the data transfer. Processors synchronize only at the reservation phase with the scalable `FetchAndAdd` and the subsequent data transfer proceeds without further synchronization. This scheme fails when pushes and pops are concurrent since the target regions are no longer non-overlapping and the subsequent data transfer cannot proceed independently. To complete the shared stack we need a mechanism to ensure that pushes and pops do not occur at the same time. This can be accomplished by creating two *rooms*, a pop room and a push room. Each processor may be in one of two rooms (or neither) but at most one of the rooms is non-empty. An increment of collection work involves the following steps in order: entering the pop room, fetching work from the shared stack, performing the work, transitioning to the push room, returning work to the shared stack. Since at most one room is non-empty, concurrent pushes and pops are avoided. Finally, this shared stack design supports scalable parallelism. Since all processors spend a bounded amount of time in the rooms, no processor waits indefinitely to enter a room.

## 3.2 Incremental and Replicating Collection

Baker proposed the first incremental collector [1]. For every unit of space that is allocated while the collector is on, the collector would copy $k$ units of space. This allowed him to bound the pause time by showing that each copy takes bounded time, and bound the memory usage by showing that the collector will make sufficient progress toward the collection. The parameter $k$ allows for a space vs. time tradeoff. The complication in designing an incremental collector is that the mutator has to work when the collection is only partially complete. Baker's algorithm deals with this by keeping a tospace invariant—the mutator can only see the copied version of objects, which are in tospace. This copy might be black or grey. If it is gray it could have pointers to objects in fromspace. Therefore, to maintain the tospace invariant, every read of a pointer by the mutator has to check which space the pointer refers to. If it refers to an object in fromspace, this object needs to be grayed and the pointer updated before proceeding. The check is called a *read-barrier* and experimentally it has been found to be quite impractical since reads are very frequent.

To avoid the use of a read-barrier, our algorithm uses a variant of the replicating collector suggested by Nettles and O'Toole [18]. Unlike Baker's algorithm, in a replicating collector the mutator can only see the original copy, which is in fromspace. While the collector executes, the mutator

continues to modify the objects in fromspace, which collectively form the *primary* memory graph, while the collector generates a *replica* memory graph in tospace. Two key invariants are crucial to the correctness of the algorithm. The *fromspace invariant* asserts that the primary graph does not refer to the replica graph, guaranteeing that the collector's operations are invisible to the mutator. The *reachability invariant* maintains that any white object is reachable from a gray object. This property is important for assuring completion of the collection as well as achieving scalable parallelism. To maintain the invariant, a write barrier is necessary. That is, whenever an object's field is modified, both the old and new object must be copied and colored gray. Since writes are significantly less frequent than reads in nearly all languages and programs, this is less of a problem than the read-barrier. This is particularly true in mostly functional languages such as SML.

Objects that are allocated while the collector is on are allocated twice, once in the primary graph and once in the replica graph. This was necessary so that when the collector finishes copying the memory graph, it can flip from the primary to the replica in bounded time.

## 3.3 Concurrency

A collector is concurrent if the program and collector can execute simultaneously. Since the program only manipulates the primary memory graph, which is not disturbed by the collector (except for the forwarding pointer field), the collector does not interfere with the mutator. On the other hand, the collector generates the replica graph by traversing the primary graph, which is being modified by the mutator (hence the name). Because a primary object might be modified after it has been copied, a replica object will not necessarily be a faithful replica unless the corresponding modification is made to the replica. A race condition arises if a primary object is being modified around the same time that it is independently copied. The following interleaving of events leads to an incorrect replica:

| Mutator | Copier | replica | primary |
|---|---|---|---|
| | read primary | – | a |
| write primary | | – | b |
| write replica | | b | b |
| | write replica | a | b |

To prevent this from occurring, the mutator's update to the replica is delayed if the modified field of the object is being copied by the collector, which is indicated by a flag in the replica object. This flag is set by the copier before it reads the primary object and is cleared after the copier modifies the replica object. Using only ordinary memory instructions, the above problematic interleaving is eliminated by forcing the mutator's replica update to follow the copier's replica write or to force the copier's primary read to follow the mutator's primary update. In the latter case, the mutator's replica update is redundant though harmless. A more detailed argument can be found in our earlier paper [2].

## 3.4 Space and Time Bounds

The key property of our previous collector were the time and space bounds. These can be summarized as follows.

- Each memory operation will take no more than $ck$ time for some constant $c$. Intuitively, $c$ is the time it takes to collect one word and $k$ is the number of words we collect per word allocated.

- If any application uses maximum reachable space R, maximum object count N, and maximum memory graph depth $D$ (the depth of a path is the sum of the sizes of objects along that path), our collector on a P-way multiprocessor requires at most $2(R(1+1.5/k)+N+5PD)$ space for any $k > 1$. For most applications, the $PD$ term is relatively small and objects are tagged so the space requirement reduces to $2R(1 + 1.5/k)$.

The proofs of these properties are contained in our previous paper, but here we outline some of the features of the algorithm required to achieve the $ck$ time bound.

**Allocation.** Multiple mutators allocate memory from fromspace. During a collection, multiple collectors allocate memory from tospace for the copied objects. To prevent conflicts, allocation is performed with a constant-time `FetchAndAdd` instruction.

**Large objects.** Since objects are not necessarily of bounded size, we cannot consider copying or scanning all the fields of an object atomically. Instead, copying and scanning of gray objects are performed field by field. To do this, we reserve a field in the replica object to indicate which fields remain to be copied. When a gray object is processed, one or more fields are processed according to the count field.

**Bounded Wait on Synchronizations.** The wait associated with the copy-copy synchronization is dependent on the time it takes to allocate space and install the forwarding pointer, both of which take constant time. The copy-write synchronization also takes constant time since the copier blocks out the writer only for the time it takes to modify one field.

**Bounded-Wait Flipping.** Turning the collector on and off is perhaps the most difficult aspect to place time bounds on. It is the only place in our algorithms in which all the processors need to synchronize and the roots need to be scanned. The size of the root set is bounded by assuming the activation records (stack) and global variables are in the heap (some run-time systems do keep the activation records in the heap anyway). Another subtle complication comes from the fact that an uninitialized large array might only be partially filled (we cannot assume that allocating a large initialized array is atomic since this would delay the synchronization). Handling a partially filled array requires keeping a counter in the array itself specifying what portion is filled.

Although such time and space bounds might not be sufficient for some real-time applications, we believe they are necessary to justify the use of the term "real-time".

# 4. MODIFIED ALGORITHM

While the algorithm presented in the previous section is scalable and concurrent with provable time and space bounds, it is impractical. The collector, as presented, has a large overhead not reflected in the bounds, and the assumptions about the globals and stack being kept in the heap are overly restrictive. In this section, we present a series of modifications that are necessary to reduce the overhead and properly handle stacks and global variables. These modifications are motivated by several cycles of experimentation.

## 4.1 Better Rooms

In our earlier rooms abstraction (section 3.1), each round of collection of work involves a processor entering the pop room, transitioning to the push room, and finally exiting. A processor remains in the pop room while fetching work from the shared stack and graying objects. This scheme has several shortcomings. The time for graying objects is considerable compared to fetching work and a processor trying to transition to the push room has to wait for all other processors already in the pop room to finish graying their objects. The wait can be significant since there may be significant variation in the time for different processors to gray objects.

These problems can be eliminated by changing to a more relaxed room abstraction in which a processor can leave the pop room rather than having to transition to the push room. Since graying objects is not related to the shared stack, it can be performed outside the rooms. This greatly reduces the time in which the pop room is active and the potential wait time of any processor. To preserve the bounded wait, each room now maintains a waiting list of processors that desire to enter. Rooms are activated in a fixed cyclic order so that overall wait time is bounded as long as no room remains occupied for an unbounded period.

## 4.2 Block Allocation

Even though `FetchAndAdd` is a scalable construct, using it for every memory allocation is inadvisable because of its expense relative to an ordinary load-add-store sequence. Furthermore, since objects are often of a size comparable to that of a cache line, allocating memory in such a fine grain fashion would split cache lines across different processors.

Instead of allocating memory directly from the shared heap each processor maintains a local pool of memory in fromspace and, when the collector is on, a local pool in tospace. Objects are allocated from the pools if the request can be satisfied. Otherwise, the local pool is discarded and a new local pool is allocated from the shared heap using a `FetchAndAdd`. Such a two-level allocation scheme has several benefits. By greatly reducing the frequency of the more expensive `FetchAndAdd`, the cost of memory allocation is reduced to the usual copying collector allocation code which is short enough that it can be inlined. Cheap allocation is important for functional languages like ML. In addition, cache behavior is improved since related objects, which are those manipulated by one processor, have improved spatial locality. Finally, local pools make it possible for the space for copying an object to be eagerly allocated before a processor is certain it will be the copier. If it should fail to be a copier, the memory is easily returned to its local pool. Without local pools, however, memory already allocated from a shared heap cannot be returned without destroying the contiguity of the unallocated space in the heap. The ability to eagerly allocate space allows the busy-wait in the copy-copy synchronization for small objects to be eliminated.

## 4.3 Write Barrier

The need for a write barrier stems from the collector's incrementality. If the $i^{th}$ field of object $x$ containing pointer value $y$ is updated with pointer value $z$, the write barrier grays the object $y$ and if $x$ has been copied, performs the corresponding update to $x$'s replica and, in so doing, grays object $z$. The code associated with these actions is substantial compared to the actual pointer update which is one instruction. Inlining this code is thus impractical. Even a function call may be unacceptable because of the disruption to the instruction cache and processor pipeline. Instead, we elect to only record the relevant information, the triple $< x, i, y >$, deferring the processing for later. The value $z$ is not necessary since it is obtainable from $x$ and $i$. Further, if the updated location contains a non-pointer value, $y$ may be omitted. This form of recording is similar to the write barrier required in a generational collector, except that a generational collector needs to record only $x + i$.

## 4.4 Reducing Double Allocation

While the collector is on, all objects that are allocated in fromspace by the mutator must be copied into tospace. This allocation barrier policy is necessary to preserve the reachability invariant. As with the write barrier, the code for this double allocation, in comparison to the normal allocation, is substantial. Deferring the double allocation is simple and does not even have an additional recording cost like the write barrier. All that is required is to replicate all objects in the current local pool whenever a new local pool is about to be allocated.

In preliminary experiments, however, we found the cost of the double allocation to be substantial. This can be seen in the following analysis. Consider an application which in steady-state has $L$ live data and is running with a fixed-sized heap of $H$. The liveness ratio is then $r = L/H$. At the start of the collection, there is $L$ live data which needs to be copied by the end of the collection. During the collection, $L/k$ additional data is allocated and hence copied. This increases the effective survival rate from $r$ to $(r + r/k)/(1 + r/k)$. For not atypical values of $r = 0.2$ and $k = 2$, the survival rate is increased from 20% to 27% which increases the collection time by 35%.

To reduce double allocation, we divide collection into two phases, non-replicating and replicating. In the first phase, we perform collection without replicating newly allocated data, so that at the end of this phase only data that was live at the beginning of the collection will have been copied. We then recompute all the roots and start a second much shorter collection during which all newly allocated objects are copied. At the end of the second phase, the replica memory graph is complete and the collection can be terminated. The 2-phase scheme greatly reduces double allocation by confining it to a short second phase. In the first phase, $L$ data is copied while $L/k$ data is allocated. Of this, $Lr/k$ is live and so we copy $Lr/k$ data plus the additional $Lr/k^2$ since we are performing double allocation. The final effective survival rate is reduced to $(r + r^2/k + r^2/k^2)/(1 + r/k + r^2/k^2)$, which, using $r = 0.2$ and $k = 2$ as before, yields 20.7%. The 2-phase algorithm requires an extra global synchronization and root set computation. However, our experiments show that this cost is more than compensated for by the reduced copying.

## 4.5 Small Objects

In our previous algorithm large and small objects were treated in the same way. Because of the need to incrementally collect large objects, small objects are also treated incrementally and copied one field at a time. This turned out to have a significant overhead. Therefore, instead of copying and scanning the fields of a small object one at a time, the entire object is locked down and the object is copied all

at once. In addition to reducing the synchronization operations, the object no longer needs to be added to and removed from the work stack nor have its tag reinterpreted for every field.

## 4.6 Globals

Most compilers assign globally accessible variables to static locations in the data segment. Because global locations are fixed at compile time, a procedure can always access them. Like registers, globals are directly accessible and so must be considered a root value when the collection starts. Similarly, a global location must be replaced with its replica value when the collection ends. Unlike the register set though, there may be arbitrarily many globals. Thus the process of turning the collector off (which involves updating global values) may take an unbounded amount of time making any real-time bounds impossible.

This problem can be solved by replicating globals like other heap-allocated object. That is, a global is actually a pair of locations, one containing the primary global value which is accessed by the mutator. The other location is available to the collector for storing the replica global value. Because this location can be modified without effecting the mutator, it can be updated long before the end of the collection. A flag is used to indicate to the mutator which of the two location holds the primary value and this flag toggled at the end of a collection when the roles of the semi-spaces are reversed. This arrangement doubles space consumption for global locations and possibly imposes a slight penalty when accessing globals. The penalty, if any, is dependent on how globals are normally compiled. In theory, since globals are already accessed indirectly through a base pointer, there should be no cost. In practice, however, taking advantage of this indirection prevents using the globals facilities already present in vendor-supplied assemblers and linkers.

## 4.7 Stacks and Stacklets

Another source of trouble concerning a timely flip comes from stacks of activation records that are used by many compilers. (Implementations that use heap-allocated read-only activation records do not have this problem.) As the stack can hold pointer values, those locations must be replaced with their replicas when the garbage collector is turned off. However, the stack can be very deep and disrupt the desired time bounds. In fact, the depth of the stack is a dynamic property and so this problem is arguably worse than that of globals.

Since we cannot wait until the end of a collection to flip all the stack slots, we must replicate the stack as the collection proceeds so that the stack work is distributed over the entire collection. Replicating the stack is, however, difficult since the mutator is continually modifying existing stack locations and pushing and popping stack frames. Instead we break up the stack into fixed-sized stacklets which are linked together with some glue code.

In this system, a deep stack becomes a string of fixed-sized stacklets, only one of which is active. While the mutator is executing in the active stacklet, the collector has a chance to replicate the other stacklets. By the time the collector is ready to turn off, only the most recent stacklet needs to be processed. Since this stacklet is bounded in size, we can also bound the processing time. During the collection, older stacklets should be processed before younger ones since younger stacklets are more likely to die.

## 4.8 Generational Collection

Generational collectors were first proposed to reduce pause times [21]. The simplest generational collector divides memory into a *nursery* and a *tenured* space. Objects are allocated from the nursery. When the nursery is exhausted, a *minor* collection is triggered in which the live objects in the nursery are copied (or *tenured*) to the tenured area. When the tenured area is exhausted after repeated minor collections, a major collection is triggered in which the live objects of the tenured area are copied to an alternate tenured area. The advantage of such an arrangement is based on the generational hypothesis which asserts that most allocated objects die shortly after they are allocated. If this is true, then a minor collection is more productive at reclaiming space then a collection in a semispace collector since a smaller fraction of objects needs to be copied. Pauses from minor collections are indeed short but the eventual major collection still makes generational collectors unsuitable for real-time applications. However, since collections are on average more productive, generational collectors are more efficient than semispace collectors when the generational hypothesis holds. In addition, they provide better data locality.

Unfortunately, adding generations is not a straightforward extension to our collector. In a non-incremental generational collector, a minor collection involves copying all live objects in the nursery (*i.e.*, those objects reachable from the roots). In the presence of mutable data structures, objects in the tenured area may refer to objects in the nursery and such references must be considered roots as well. Since the collector is incremental, these tenured references may not be modified during the collection since the mutator is executing. Instead the referenced objects are copied during the collection and only at the end of the collection are the references themselves modified. In general, the number of references is unbounded and all the references cannot be atomically modified without breaking the real-time property.

The solution is to use two fields for each mutable pointer field. During any particular collection, one field is used by the mutator while the other field is updated by the collector. The roles of the fields are reversed at the end of each collection. Our initial experiments show that this scheme allows the same real-time bounds to be met while giving most of the overall performance improvement of using generations. Further, the benchmarks show that even a modestly-sized array of 10,000 pointer values was enough to make these modifications necessary. The major drawback to this scheme is that field access by the mutator is slightly more expensive and more space is required for mutable pointer fields.

# 5. EVALUATION

## 5.1 Implementation Description

To evaluate the algorithms, we implemented several collectors within the runtime system [4] for the TILT SML compiler. SML is a statically typed, functional language with a module system. The prevalence of closures, algebraic datatypes, and tupling in most SML programs leads to a high allocation rate, providing a tough challenge for a collector. Further, the TILT compiler uses both global variables and stacks and thus requires the modifications described in sections 4.6 and 4.7.

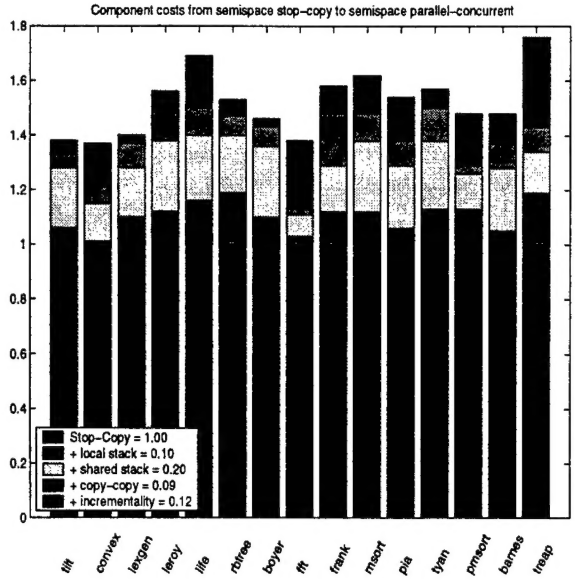



**Figure 1: Time costs of parallelism and incrementality. The cost of the various mechanisms are displayed relative to the cost of a non-parallel, non-incremental collector which is normalized to 1.00.**

The collector code consists of approximately 6000 lines of C code and 500 lines of assembly code (mostly glue code). Altogether, we implemented 6 collectors: semispace stop-copy, semispace parallel, semispace concurrent, generational stop-copy, generational parallel, and generational concurrent. The gcc compiler with -O2 was used for compilation. Critical functions, determined with profiling, are inlined. Currently the TIL SML compiler as well as the runtime system and garbage collector runs on Alpha workstations running Digital UNIX and on UltraSparc-II workstations running Solaris.

Our experiments were performed on a 6-processor and a 64-processor UltraSparc-II workstation. Experiments were performed multiple times and on both machines to assure that the variance in timing was low. Interestingly, these two machines exhibited slightly but consistently different performance characteristics, probably because the machines have different memory subsystems. The data presented in this section comes from executions from the 64-processor machine.

To obtain reasonable real-time behavior, the runtime system locks down all pages that are part of the heap at the beginning of execution. Even so, context switches or scheduling glitches can disrupt our timings. Because Solaris is not a real-time operating system, we can only cope with this by running at a higher priority and performing the experiments when the machine is not heavily loaded.

## 5.2 Benchmarks

The experiments were conducted on 15 benchmarks, some of which are standard for SML [20]. They include symbolic processing (life, knuth-bendix, boyer-moore, grobner polynomials, lexgen, tree search with backtracking), numerical applications (fft, pia), large application (the TIL compiler itself), data intensive (merge sort, red-black tree), and parallel applications (convex-hull, barnes-hut, treap manipulation). Some of the benchmarks have large arrays (convex-hull and fft), deep stacks (knuth-bendix), and high mutation rates (tree search, convex-hull, and fft).

## 5.3 Cost of Parallelism and Incrementality

To understand the overhead of using a parallel and/or incremental collector, we analyze the costs of various necessary mechanisms by measuring the cost of running a collector with various features enabled. The basis for our comparison is a non-parallel, non-incremental semispace collector which does not have an explicit data structure for maintaining the set of gray objects. We successively take steps toward a parallel and incremental/concurrent collector by adding, in order, an explicit local stack, a shared stack, copy-copy synchronization, and incrementality.

Figure 1 illustrates the relative time cost of these components with the cost of the base collector normalized to 1. On average, the use of an explicit local stack instead of the implicit Cheney queue costs 10%. The copy-copy synchronization is necessary for parallelism, requiring 9%. Finally, load-balancing (room synchronization and communication between the local stacks and the shared stack) requires an additional 20%. Overall, the cost of scalable parallelism is 39%. If incrementality is required, an additional 12% overhead is incurred due to double allocation, more frequent context switches, and the write barrier. For each benchmark the total memory available was kept constant across the different collector variants. This means that part of the overhead of the incrementality is due to slightly more frequent collections.

Because of the use of a write barrier, one might wonder how efficient a concurrent collector would be in the context of a language that is more heavily based on mutation, such as Java. We don't have conclusive evidence to answer this question, but we have noticed that in the applications with a substanaial amount of mutations (frank, convex-hull, and fft), the cost of the write barrier was not significant.

## 5.4 Utilization and Maximum Pause Time

As discussed in Section 2.3, maximum pause time alone is too limited for quantifying the degree to which a garbage collector is real-time or incremental. It fails to take into account whether the mutator has reasonable access to the processor and is able to make sufficient progress towards its task. Instead we defined the more general notion of minimum mutator utilization (MMU).

Figure 2 shows the MMU of all benchmarks for the stop-copy collector and the parallel, concurrent collector for two values of $k$ (the rate of collection). The minimum mutator utilization is linearly plotted against time-granularity which is shown logarithmically. As expected, the utilization curves generally increase as the granularity increases (though it is not strictly monotone). At the low end, the utilization falls to zero when the granularity is below the longest collection for the stop-copy collector or, for the incremental collector, below the longest increment of work. Naturally, this point of minimum granularity is lower for an incremental collector. At the high end of granularity, the stop-collector provides greater utilization than an incremental collector since the incremental collector has an overhead compared to the stop-copy collector. There is a range in which, if utilization is of concern as in a real-time collector, the incremental collector is preferable. On the low end, this range is from 3 ms to 10

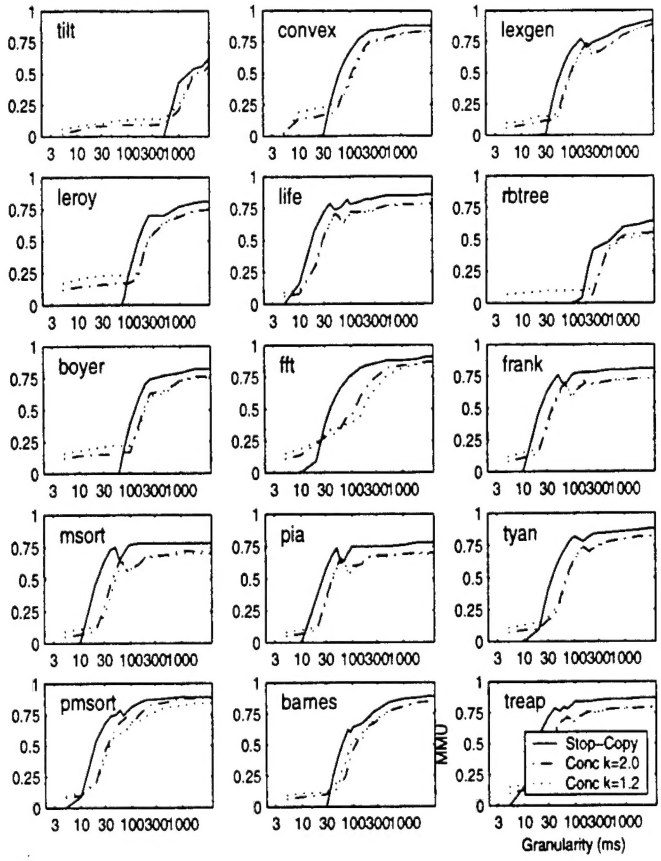



**Figure 2: MMU vs granularity (ms) of semispace collectors: non-incremental, incremental (k=1.2), and incremental (k=2.0)**

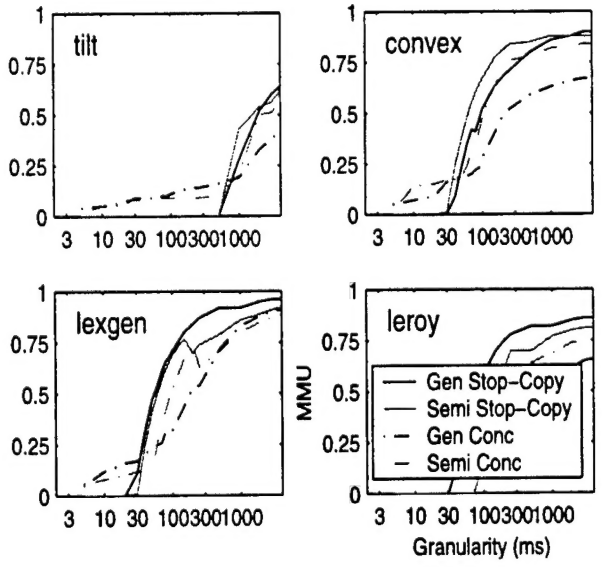



**Figure 3: MMU vs granularity (ms) of 4 collectors. Generational collectors are marked with darker lines. Independently, the line for the incremental collectors are dashed.**

ms but for more memory-intensive applications this range is up to almost 1 s.

For each benchmark the total memory available is held constnt the same for all three collector variants. The parameter $k$ controls the rate of collection and effects the utilization level. Recall that $k$ is the number of units of space collected for each unit allocated. When $k$ is low, less collection work is performed relative to allocation and so the collector runs less, leading to higher utilization. However, a lower value of $k$ can potentially require more memory. Given that we keep the available memory fixed, lowering $k$ translates to more frequent collection and potentially lower overall performance.

In summary, at the 10 ms granularity, the incremental, concurrent collector provides a minimum utilization of about 10% for $k = 2.0$ and 15% for $k = 1.2$. On the other hand, the stop-copy collector provides 0% utilization at 10 ms at 12 of the 15 benchmarks.

## 5.5 Scalability

We consider two types of benchmarks for testing scalability. The first class consists of the benchmarks that are already (finely) parallel using a fork-join construct (convex-hull, barnes-hut, and treap). The remaining cases are coarsely parallel programs in which there are as many threads as processors, each thread running a sequence of non-parallel benchmarks in a different order.

Figures 4 and 5 show how the mutator and collector times scale for the benchmarks. Each benchmark is run with and without load-balancing and from 1 to 8 processors. Each bar on the graphs represents the sum of the times across the processors. This time is divided into four states: running application, idle from no application work, garbage collecting, and idle from lack of collection work. Perfect scalability is achieved when there is no idle time, and the time spent in the mutator and collector does not increase with the number of processors. The execution times have been normalized to the one processor time. In the case of the coarsely parallel benchmarks, the workload is proportional to the number of threads so we report execution time divided by the number of threads.

For the finely parallel benchmarks, the normalized time increases from about 0.2 at 1 processor to 0.3 at 8 processors while the mutator increases from 0.8 at 1 processor to 1.06 at 8 processors. The idle time of the mutator greatly varies depending on the parallel application and the number of processors, from 0.0 to 0.8. We note that this idle time has to do with the application and thread scheduler and little to do with the collector. For the garbage collector, the idle time is under 0.1 when work sharing is enabled. Without work sharing, the collector spends 0.4 time in an idle state. For these benchmarks, the collector's scalability is about 5.5.

The coarsely parallel benchmarks exhibit different behavior. There is generally little mutator idle time and what is present is compensated by a decrease in the mutator time. This indicates that some threads finish before others even though each is doing the same amount of work. Up to 8 processors, the garbage collector scales linearly and even superlinearly in some cases. When load balancing is enabled, there is almost no idle time. However, without load balancing, there is a significant idle time of 0.32 which, in some cases, exceeds the collection time itself. For these benchmarks, the collector's scalability is about 7.8.

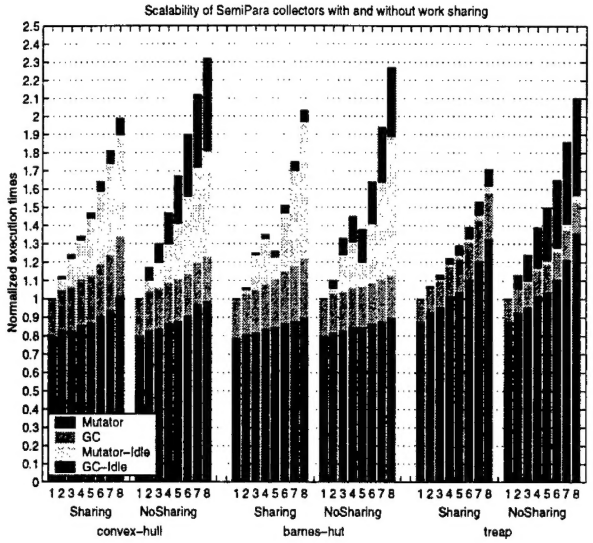



**Figure 4: Total processor time spent in various states for coarsely parallel benchmarks.**

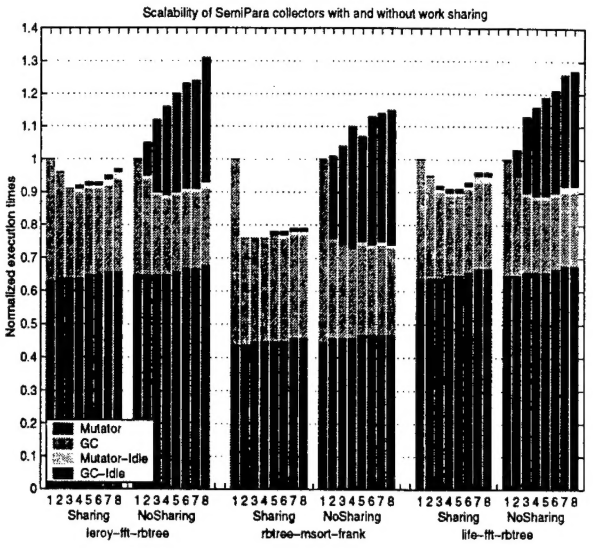



**Figure 5: Total processor time spent in various states for finely parallel benchmarks.**

We conclude that our collector scales well up to 8 processors and that load-balancing is critical to achieving this.

## 5.6 Generations

Next, we examine the effect of generations on utilization level. Figure 3 shows the MMU curve of 4 collectors which are differentiated by whether they are generational and/or incremental. Generally, the presence of generations do not greatly affect the MMU curve. On average, in the non-incremental collectors, generations only slightly lower the maximum pause time. Clearly, to gain significant responsiveness, adding generations is insufficient and incrementality is necessary.

We also note that although it appears from the diagrams that the generational concurrent collector is often less efficient than the semispace version, this is actually not the case. Because of the locality effects of the generational collector, the mutator runs significantly faster, so the overall time is faster although the utilization is not any better, and in some cases worse.

# 6. RELATED WORK

Concurrent garbage collection was independently introduced by Steele [19] and Dijkstra [5], both of whom based their work on mark-and-sweep collectors. By extending Cheney's copying collector [3] with a read barrier, Baker proposed the first real-time copying collector with bounds on memory use and time [1]. He proved that if each allocation copied $k$ locations then his collector would require only $2(R(1 + 1/k))$ locations. He also suggested how it could be extended to arbitrarily sized structures, although this required both a read and write barrier and also an extra link-word for every structure.

As part of the Multilisp system, Halstead developed a multiple-mutator multiple-collector variant of Baker's basic algorithm for shared memory multiprocessors [14]. In their scheme, every processor maintains its own from- and to-space and traces its own root set to move objects from any from-space to its own to-space. This algorithm has several properties that make it neither time nor space efficient, both in theory and practice. First, the separation of space among the processors can lead to one processor running out of space, while others have plenty left over. In addition to making it unlikely that any space bounds stronger than $2RP$ can be shown (*i.e.*, a factor of $P$ more memory than the sequential version), Halstead noted that this problem actually occurred in practice. A related problem is that since each processor does its own collection from its own root set without sharing the work, one processor could end up doing much more copying than the others. Since no processor can discard its from-space until all processors have completed scanning, all processors will have to wait while the one processor completes its unfair share of the collection.

Herlihy and Moss [15] described a lock-free variant of the Multilisp algorithm. The locks are avoided by keeping multiple versions of every object in a linked list. Although the algorithm is lock-free it cannot make guarantees about space or time. For example, to read or write an object might require tracing down an arbitrarily long linked list.

Doligez and Gonthier describe a multiple mutator single collector algorithm [7]. They give no bounds on either time or space. In fact the collector can generate garbage faster than it collects it, requiring unbounded memory. More generally, no single collector algorithm can scale beyond a few processors since one processor cannot keep up with an arbitrary number number of mutators. On the other hand the Doligez-Gonthier algorithm has some properties—including minimal synchronizations and no overheads on reads—that make it likely to be practical on small multiprocessors. Domani et. al. describe a generational version of the collector [9], but it has the same scalability issues as the original.

Our replicating algorithm is loosely based on the replicating scheme of Nettles and O'Toole [18]. Beyond the basic idea of replication, however, there are few similarities. This is largely due to the fact that they do not consider multiple collector threads and do not incrementally collect large structures.

Incremental or real-time collectors are based on a variety

of techniques, leading to varying worst-case pause times: 2 - 5 ms [11], 15 ms [16], 50 ms [18], and 360 ms [8]. However, direct comparison of worst-case pause times without considering mutator access to the processor can be misleading.

## 7. CONCLUSION

We have presented an implementation of a scalably parallel, concurrent, real-time garbage collector. The collector allows for multiple collector and mutator threads to run concurrently with minimal thread synchronization. We believe that our modified algorithm maintains the time and space bounds of our previous theoretical algorithm, which gives some justification for calling the algorithm "real-time". Our experiments and the analysis of the MMU across 15 benchmarks gives further justification.

## 8. ACKNOWLEDGEMENTS

Thanks to Toshio Endo and the Yonezawa Laboratory for use of their 64-way Sparc multiprocessor. Thanks to the general support of Robert Harper and the FOX project. We are grateful to Doug Baker for his comments.

## 9. REFERENCES

[1] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[2] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

[3] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[4] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[5] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46.* Springer-Verlag, New York, 1976.

[6] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[7] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[8] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[9] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of 2000 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 274–284, May 2000.

[10] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Master's thesis, University of Tokyo, February 1998.

[11] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real time constraints. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[12] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[13] Robert H. Halstead. Multiple-processor implementations of message passing systems. Technical Report TR-198, MIT Laboratory for Computer Science, April 1978.

[14] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[15] Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), May 1992.

[16] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 1–9, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[17] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[18] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[19] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[20] Tarditi, Morrisett, Cheng, Stone, Harper, and Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Programming Languages Design and Implementation (PLDI)*, pages 181–192, 1996.

[21] David M. Ungar and David A. Patterson. Berkeley Smalltalk: Who knows where the time goes? In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 189–206. Addison-Wesley, 1983.